Reliable Software Systems

Week 8: Scalable Design Patterns

February 28, 2018
Alyssa Pittman
University of Washington Allen School

Motivating Example: young Foursquare

In 2010, Foursquare was growing at a rate of half a million users per month.

They had recently split their data into two database shards.

One shard filled more quickly than the other, and when it exceeded RAM on the server, performance ground to a halt.

They added another shard and moved data to it, but due to data fragmentation, the existing shard still performed poorly.

Compaction took too long, but after downtime (11 hours total) and a full restore of data from a backup, all the shards performed well again.

https://groups.google.com/forum/#!topic/mongodb-user/UoqU8ofp134

Architecture

These are *system* design patterns rather than *software* design patterns

Tradeoffs between approaches:

- Scalability can be more complicated than the less-scalable approaches
- There are times and places to use the less-scalable approaches

Three main ideas

Distribution: e.g. one server isn’t enough to deal with all the load

Caching: e.g. reducing load on the data storage system

Asynchronous processing: e.g. work takes too long to do all at once

Use case: receiving updates

Less-scalable pattern: polling, using transactions

More-scalable pattern: queues, messages

Queues

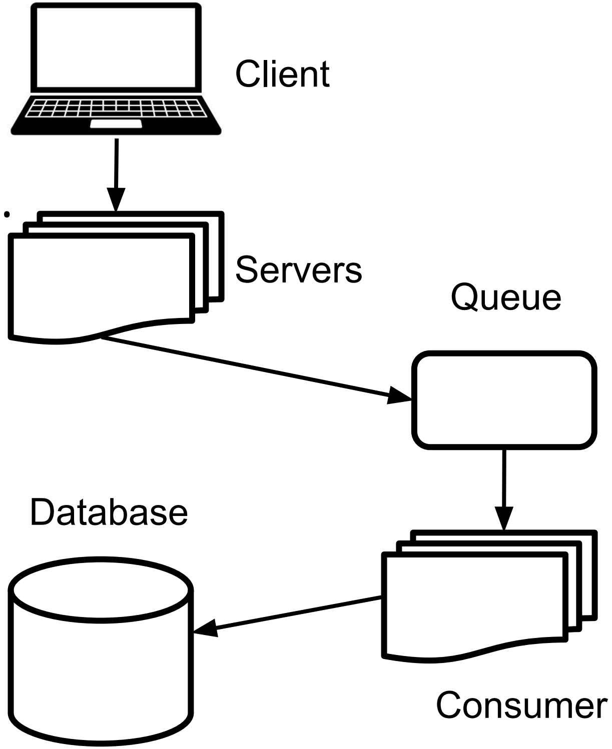
Queue work so that a consumer can process it later.

Pros:

- Frees application servers for serving
- Distributes work over consuming servers
- Asynchronous nature smooths out spikes

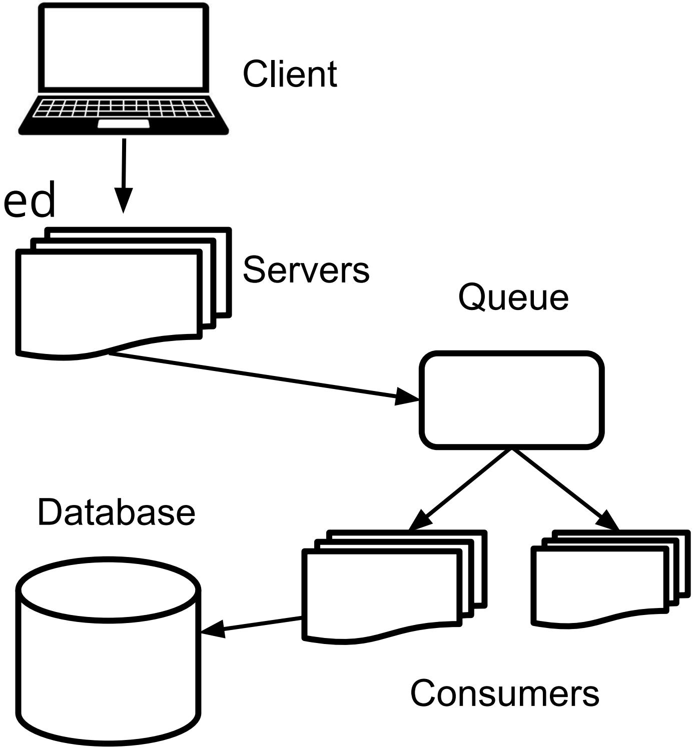
Cons:

- Work not done immediately
- At-least-once messaging - message retries? Out-of-order message arrival?



Messages

Publish messages to consumers who have subscribed (like queues, but one event can be processed by multiple subscribers)



Pros:

Loose coupling of all consumers

Use case: scaling data usage

Less scalable: add more resources to a database server

More scalable: sharding, caching, scalable databases

Sharding

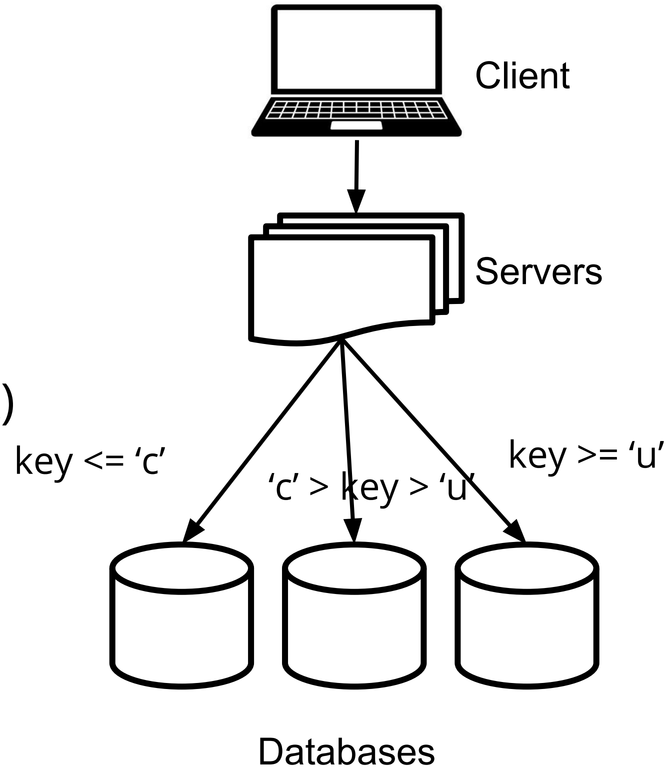
Split data into multiple databases

Pros:

- Horizontally scales database (distributes work)

Cons:

- Rethink the schema if multiple pieces of data need to be accessed at once
- Still have single points of failure
- Shards can become unbalanced



Caching

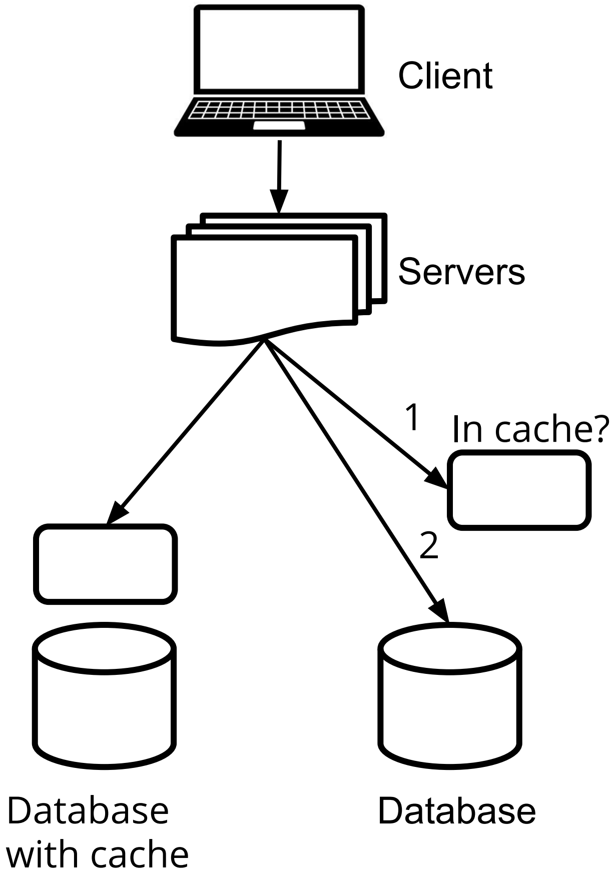
Precalculate results or store frequently-used results (database-level or application-level caching)

Pros:

- Expensive work done asynchronously
- Amount of work reduced

Cons:

- Cache invalidation is difficult



Scalable databases

Based on the *structure* of your data (schema or json blob?),
your *retrieval patterns* (sequential access or key based access?), and
consistency requirements, pick a NoSQL database solution.

Pros:

Relaxes consistency to give better availability & partition tolerance

Cons:

Joins and queries often need to be done by application

Application needs to deal with potentially inconsistent data

Use case: processing lots of data

Less scalable: run a batch job on a single server

More scalable: Map/Reduce, streaming

Map/Reduce

Store data in a distributed file system, use Map/Reduce to process it.

Pros:

Good for I/O bound tasks

Distributed use of resources

Cons:

Still a batch/offline job

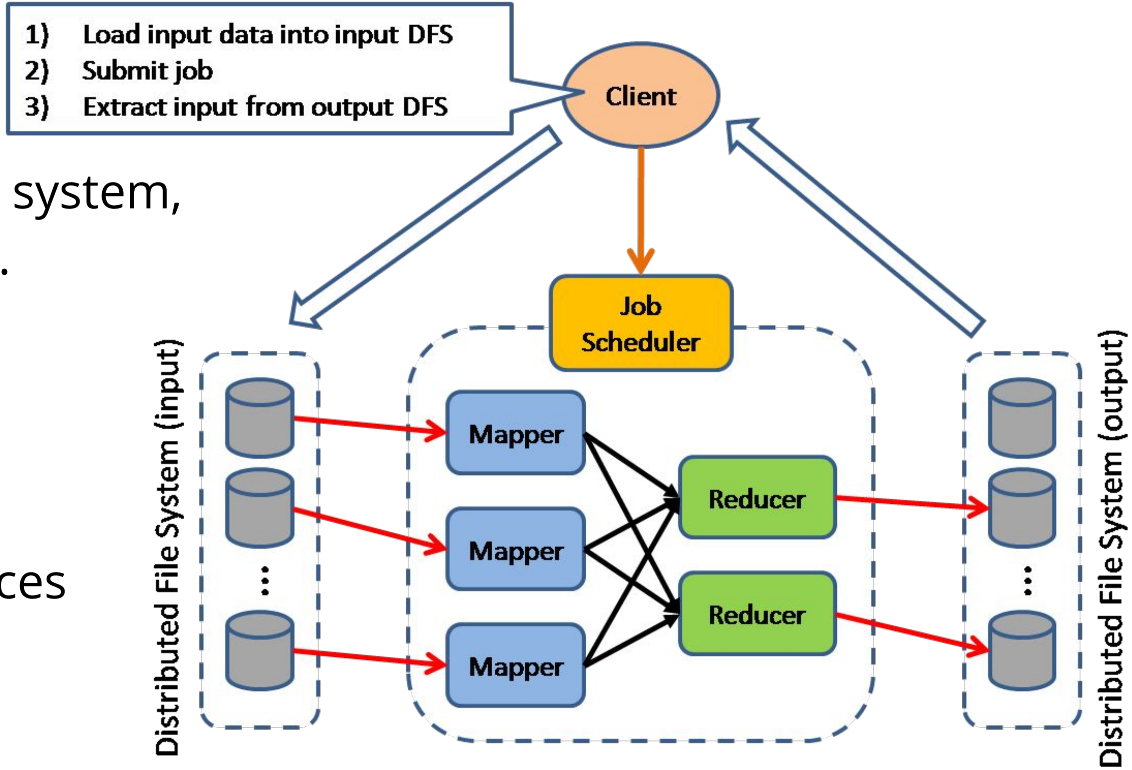

1) Load input data into input DFS
2) Submit job
3) Extract input from output DFS
Client
Job Scheduler
Distributed File System (input)
Mapper
Mapper
Mapper
Reducer
Reducer
Distributed File System (output)
...
...

Image from http://horicky.blogspot.com/2010/10/scalable-system-design-patterns.html

Streaming

Process data as it arrives by writing “agents” to process each event

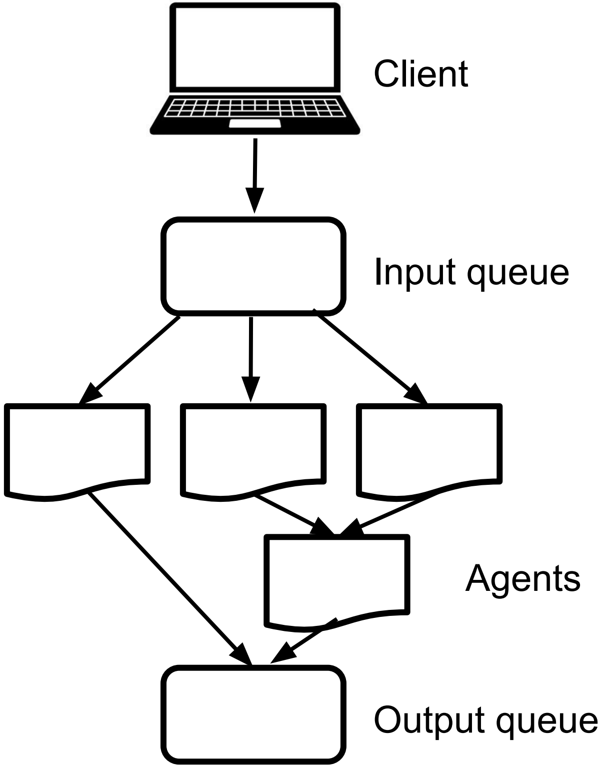
Pros:

Immediately processes data

Cons:

Often approximates outputs

Can only do a single pass over data



END